Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 9 and Homework 9: next Monday 11/21 @ 11:59 pm
  - Assignment 3: ~~Monday 11/28~~ Friday 12/9 @ 11:59 pm
  - Assignment 4: Friday 12/9 @ 11:59 pm

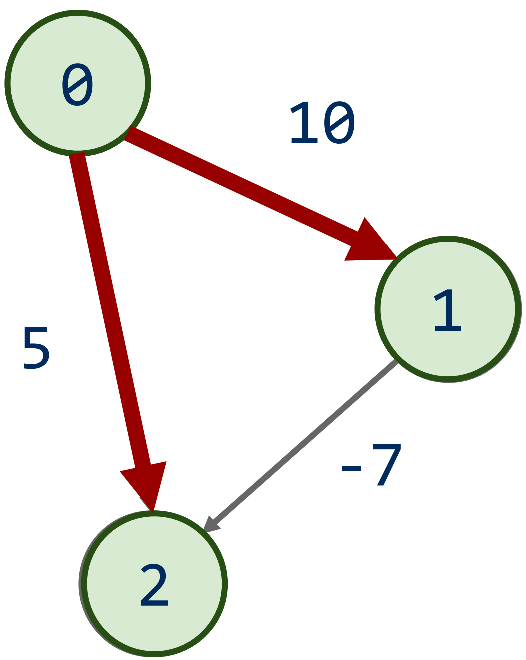
Previous lecture

- Weighted Shortest Paths problem
  - Dijkstra’s shortest paths algorithm
  - Bellman-Ford’s shortest paths algorithm

This Lecture

- Dynamic Programming

Dijkstra's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	5	0

Incorrect!

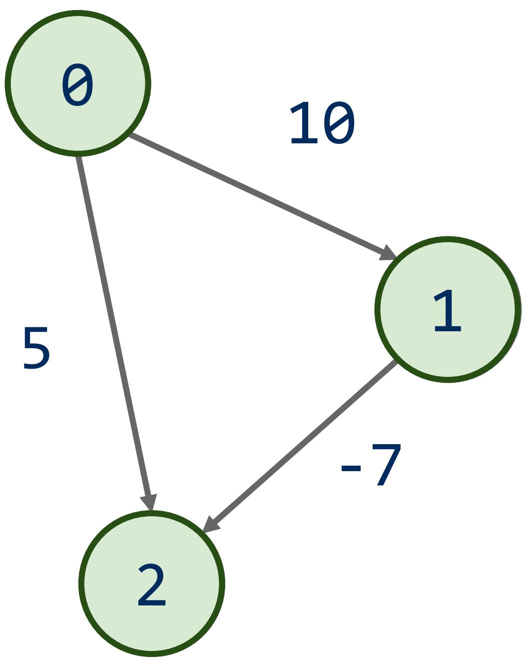
Analysis of Dijkstra’s algorithm

Dijkstra’s is correct only when all edge weights >= 0

Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty:
    - cur = pop a vertex from Q
    - For each non-parent neighbor x of cur:
        - Compute distance from start to x through cur
            - distance[cur] + weight of edge between cur and x
        - if computed distance < distance[x]
            - Update distance[x]
            - add x to Q if not already there

Bellman-Ford's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	3	1

FIFO Q:

0
1
2

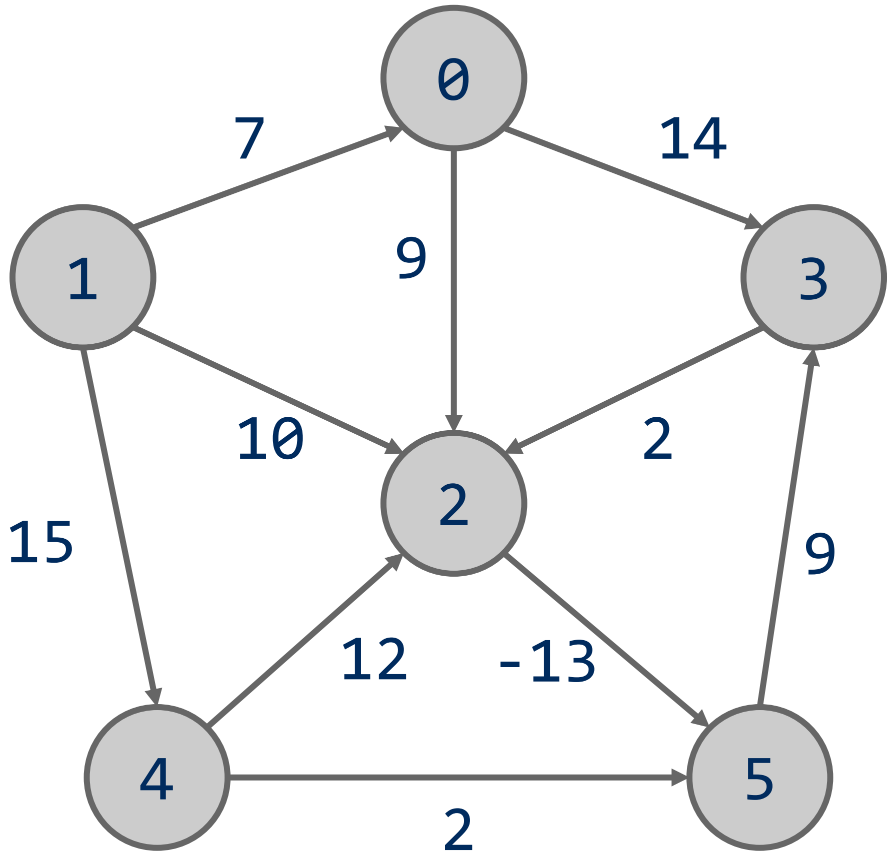
Correct!

Analysis of Bellman-Ford's algorithm

Bellman-Ford's is correct even when there are negative edge weights in the graph but what about negative cycles?

- a negative cycle is a cycle with a negative total weight

Bellman-Ford’s example with a negative cycle



Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty **and no negative cycle has been detected**:
    - cur = pop a vertex from Q
    - For each non-parent neighbor x of cur:
        - Compute distance from start to x through cur
            - distance[cur] + weight of edge between cur and x
        - if computed distance < distance[x]
            - Update distance[x]
            - add x to Q if not already there
    - check for a negative cycle in the current Spanning Tree every v edges

Let’s change focus into a different type of problems

- We will get back to graphs after the break!

Consider the change making problem

- What is the minimum number of coins needed to make up a given value k?
- If you were working as a cashier, what would your algorithm be to solve this problem?

This is a *greedy algorithm*

- At each step, the algorithm makes the choice that seems to be best at the moment
- Have we seen greedy algorithms already this term?
  - Yes!
    - Building Huffman trees
    - Nearest neighbor approach to travelling salesman

… But wait …

- Nearest neighbor doesn't solve travelling salesman
  - Does not produce an optimal result
- Does our change making algorithm solve the change making problem?
  - For US currency…
  - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?

So what changed about the problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices
- Why is optimal substructure not enough?

Finding all subproblems solutions can be inefficient

- Consider computing the Fibonacci sequence:

```
int fib(n) {
        if (n == 0) { return 0 };
        else if (n == 1) { return 1 };
        else {
                return fib(n - 1) + fib(n - 2);
        }
}
```

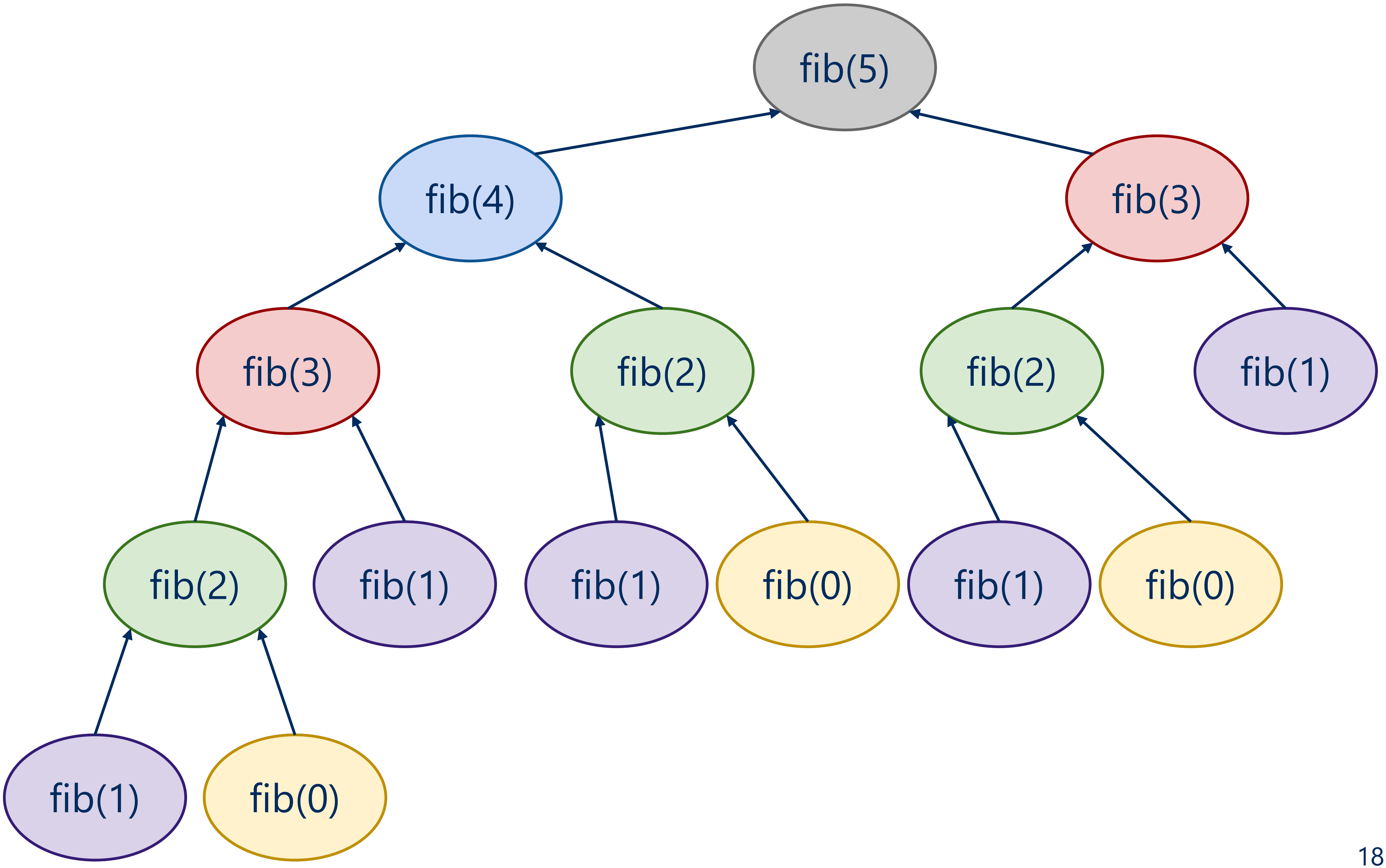
- What does the call tree for n = 5 look like?

+

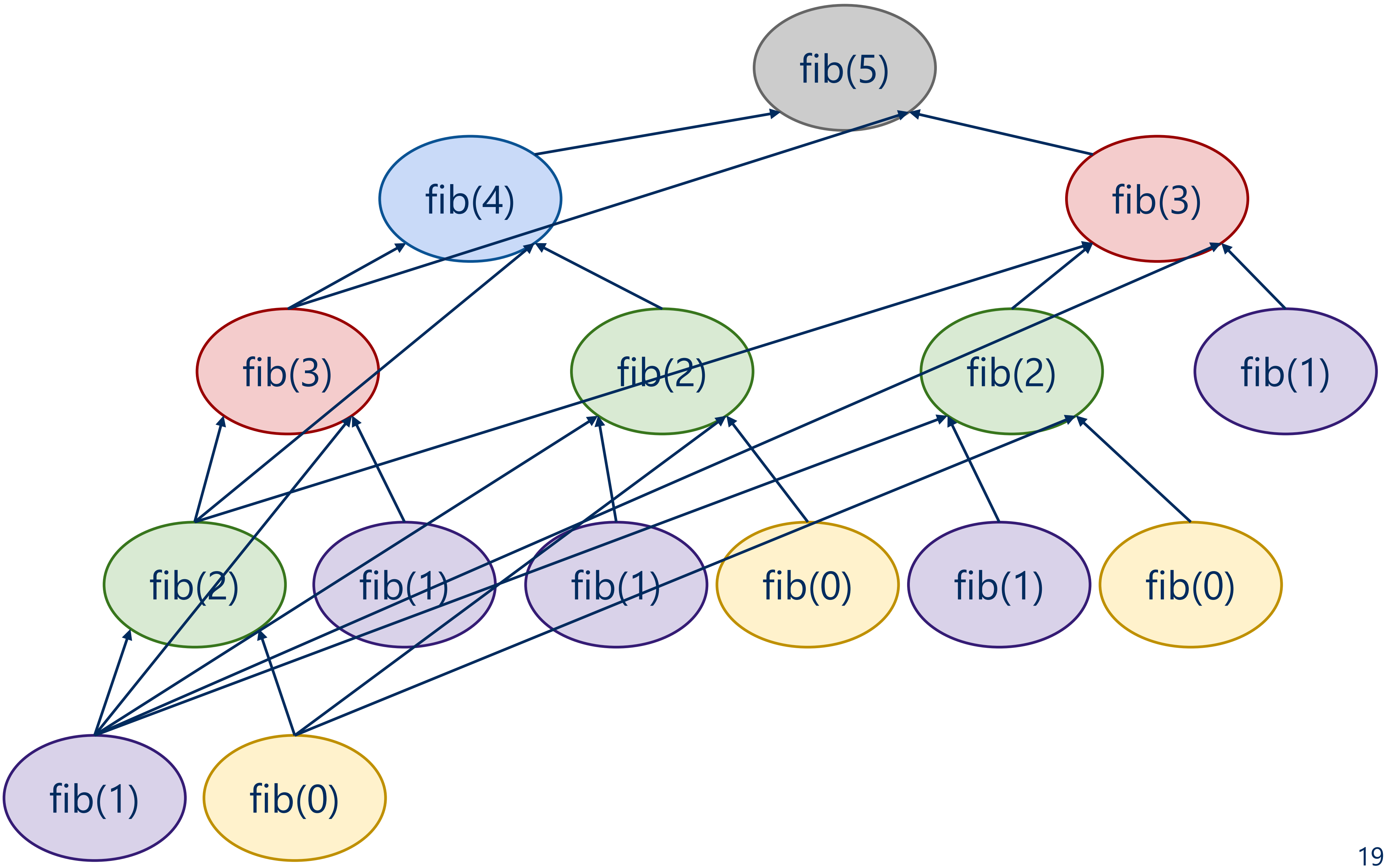
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

+

fib(5)


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

How do we improve?



Memoization

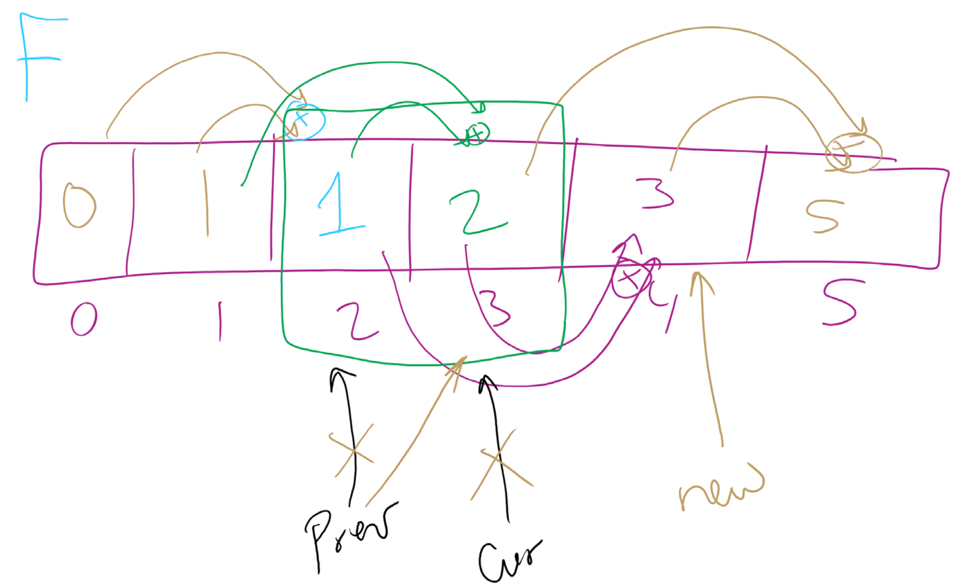
```
int[] F = new int[n+1];
   F[0] = 0;
   F[1] = 1;
   for(int i = 2; i <= n; i++) { F[i] = -1 };

int dp_fib(x) {
         if (F[x] == -1) {
               F[x] = dp_fib(x-1) + dp_fib(x-2);
         }
         return F[x];
   }
```

Note that we can also do this bottom-up

```
int bottomup_fib(n) {
    if (n == 0)
        return 0;
    int[] F = new int[n+1];
    F[0] = 0;
    F[1] = 1;
    for(int i = 2; i <= n; i++) {
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
      }
```

Can we improve this bottom-up approach?

```
int improve_bottomup_fib(n) {
    int prev = 0;
    int cur = 1;
    int new;
    for (int i = 0; i < n; i++) {
        new = prev + cur;
        prev = cur;
        cur = new;
    }
    return cur;
}
```



Where can we apply dynamic programming?

- To problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times

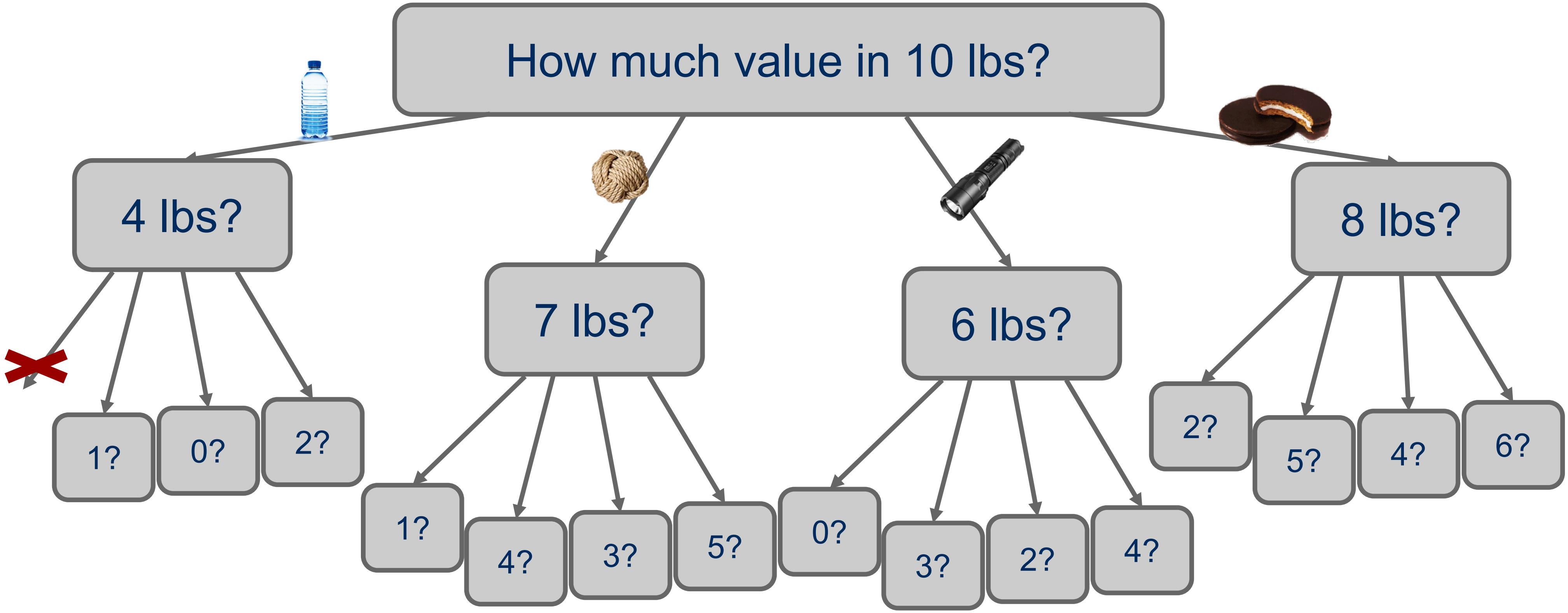
Problem of the Day Part 3: The unbounded knapsack problem

- Given a knapsack that can hold a weight limit L, and a set of n types items that each has a weight (w_i) and value (v_i), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?

Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

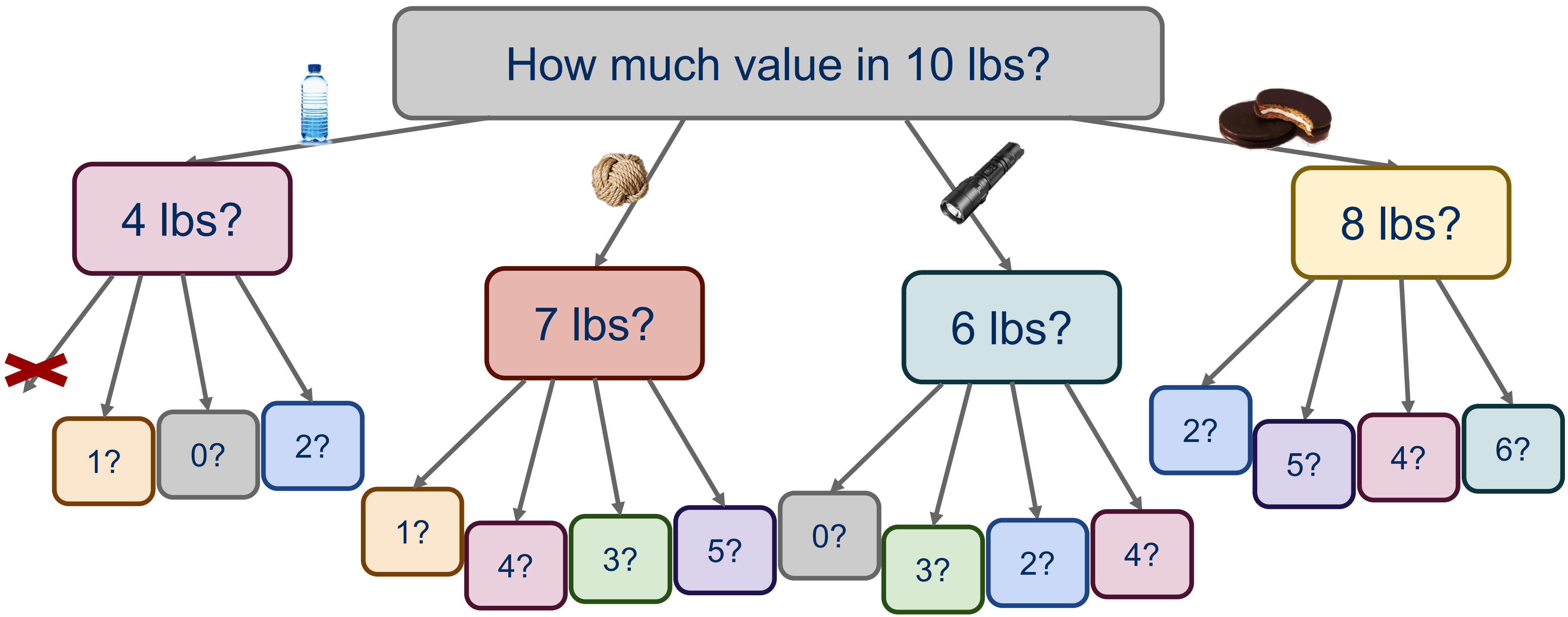
10 lb.
capacity



Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Bottom-up Solution

				
weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

Bottom-up solution

```
K[0] = 0
for (l = 1; l <= L; l++) {
      int max = 0;
      for (i = 0; i < n; i++) {
            if (w_i <= l && v_i + K[l - w_i]) > max) {
                  max = v_i + K[l - w_i];
            }
      }
      K[l] = max;
}
```

What would have happened with a *greedy* approach?

- At each step, the algorithm makes the choice that seems to be best at the moment
- Have we seen greedy algorithms already this term?
  - Yes!
    - Building Huffman trees
    - Prim’s, Kruskal’s MST
    - Dijkstra’s Single-Source Shortest Paths

The *greedy algorithm*

- Try adding as many copies of highest value per pound item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moonpie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb knapsack?
  - 44
    - Bogus!

But why doesn't the greedy algorithm work for this problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices
- Why is optimal substructure not enough?

The bottom-up approach is called dynamic programming!

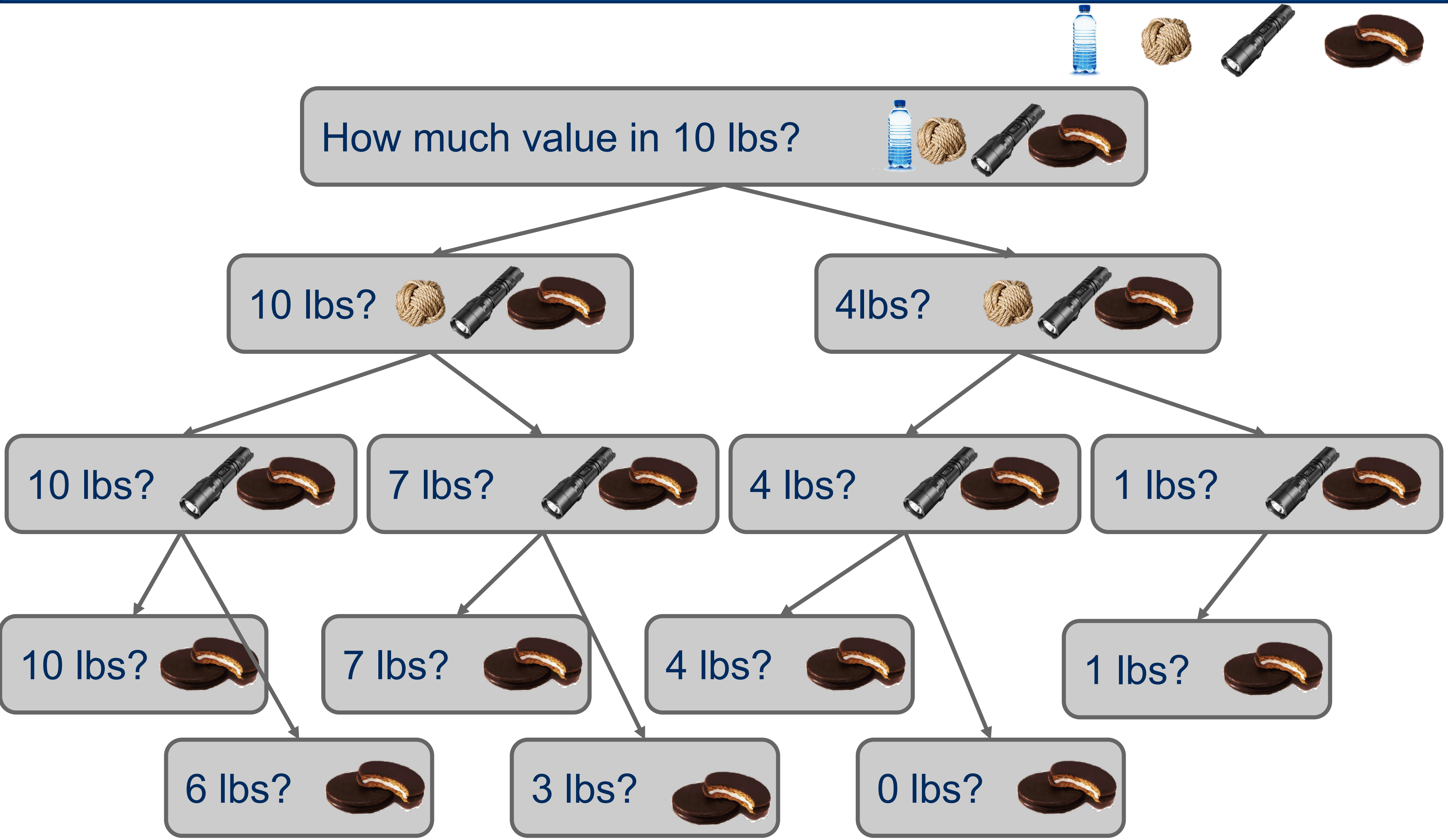
- Applies to problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times
- Greedy Choice Property is not required

Dynamic Programming Example 1: The 0/1 knapsack problem

- What if we have a finite set of items that each has a weight and value?
  - Two choices for each item:
    - Goes in the knapsack
    - Is left out

0/1 Recursive solution

weight:	6	3	4	2
value:	30	14	16	9



Recursive solution

```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0 };
    if (wt[n-1] > L) {
        return knapSack(wt, val, L, n-1)
    }
    else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                                knapSack(wt, val, L, n-1)
                               );
    }
}
```

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
K[n+1][L+1]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,   3,   4,   2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0								
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16						
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0									

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0	9	9	16	16	30	30	39	44	46

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                                          K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```

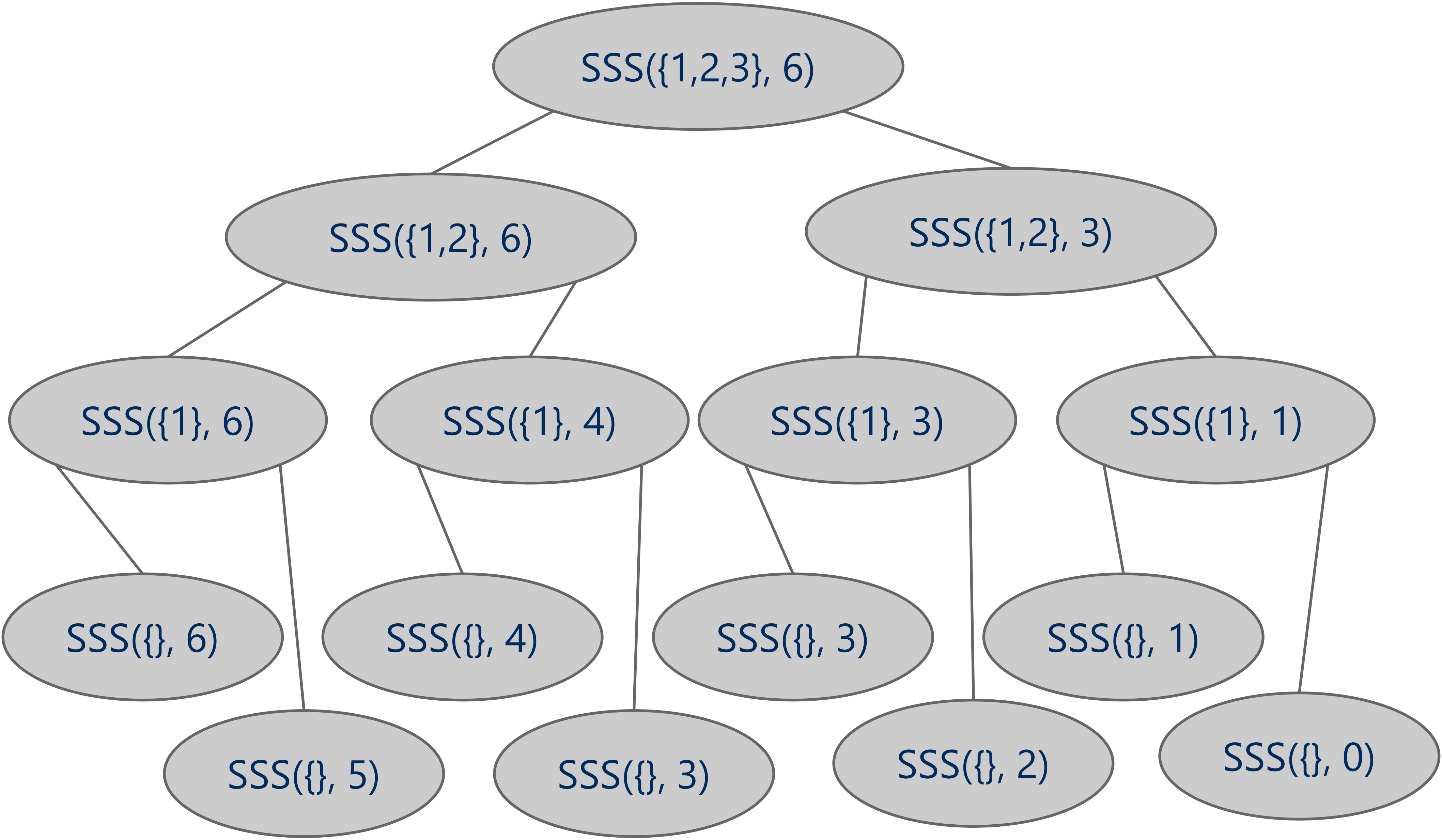
To review...

- Questions to ask in finding dynamic programming solutions:
  - Does the problem have optimal substructure?
    - Can solve the problem by splitting it into smaller problems?
    - Can you identify subproblems that build up to a solution?
  - Does the problem have overlapping subproblems?
    - Where would you find yourself recomputing values?
      - How can you save and reuse these values?

Dynamic Programming Example 2: Subset sum

- Given a set of non-negative integers S and a value k, is there a subset of S that sums to exactly k?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {
    if (sum == 0)
          return true;
    if (sum != 0 && n == 0)
          return false;
    if (set[n-1] > sum)
          return SSS(set, sum, n-1);
    return SSS(set, sum, n-1)
          || SSS(set, sum-set[n-1], n-1);
}
```

- What would a dynamic programming table look like?

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[sum+1][n+1];
    for (int i = 0; i <= n; i++) subset[0][i] = true;
    for (int i = 1; i <= sum; i++) subset[i][0] = false;
    for (int i = 1; i <= sum; i++) {
       for (int j = 1; j <= n; j++) {
              subset[i][j] = subset[i][j-1];
              if (i >= set[j-1])
                     subset[i][j] ||= subset[i - set[j-1]][j-1];
       }
    }
    return subset[sum][n];
}
```

Example 3: Change making problem

Consider a currency with n different denominations of coins d_1, d_2, ..., d_n. What is the minimum number of coins needed to make up a given value k?

Solution Attempt

If you were working as a cashier, what would your algorithm be to solve this problem?

... But wait ...

- Does our greedy change making algorithm solve the change making problem?
    - For US currency...
    - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
        - What denominations would it pick for k=6?

So, how can we solve the change making problem optimally?

We will see a dynamic programming algorithm in the recitation of this week.